S-GRID: A Versatile Approach to Efficient Query Processing in Spatial Networks

Xuegang Huang, Christian S. Jensen, Hua Lu, and Simonas Šaltenis

Department of Computer Science, Aalborg University
Fredrik Bajers Vej 7E, DK-9220, Aalborg, Denmark
{xghuang,csj,luhua,simas}@cs.aau.dk

Abstract. Mobile services is emerging as an important application area for spatio-temporal database management technologies. Service users are often constrained to a spatial network, e.g., a road network, through which points of interest, termed data points, are accessible. Queries that implement services will often concern data points of some specific type, e.g., Thai restaurants or art museums. As a result, the relatively few data points are relevant to a query in comparison to the number of network edges, meaning that queries, e.g., k nearest-neighbor queries, must access large portions of the network.

Existing query processing techniques pre-compute distances between data points and network vertices for improving the performance. However, pre-computation becomes problematic when the network or data points must be updated, possibly concurrently with the querying; and if the data points are moving, the existing techniques are inapplicable. In addition, multiple pre-computed structures must be maintained—one for each type of data point. We propose a versatile pre-computation approach for spatial network data. This approach uses a grid for pre-computing a simplified network. The above-mentioned shortcomings are avoided by making the pre-computed data independent of the data points. Empirical performance studies show that the structure is competitive with respect to the existing, more specialized techniques.

1 Introduction

In step with the emergence of an infrastructure that enables the deployment of mobile services, the database research community has begun to consider the challenges brought on by scenarios where services are delivered to large populations of mobile users. In one important setting, the service users are constrained to a spatial network such as a road network, and the services involve nearest-neighbor queries on points of interest, which we term data points, that are accessible via the network.

As an example, consider Figure 1 where we aim to find the nearest restaurant for a mobile user q among six restaurants $\mathrm{R}_1, \mathrm{R}_2, \ldots, \mathrm{R}_6$. To address this type of problem, we model the spatial network as a graph structure. Specifically, a spatial network is modeled as a directed and labeled spatial graph $RN = (V, E)$, where $V = \{v_0, v_1, \ldots, v_m\}$ is a finite set of vertices and E is a finite set of edges. Vertices model intersections and starts and ends of roads, and each vertex has an associated point position in two-dimensional Euclidean space.

An edge models the part of a road in-between two vertices. It is a three-tuple $e_{i,j} = (v_i, v_j, l)$, where $v_i, v_j \in V$ is the start and end vertex of the edge, respectively, and l captures the travel length of the edge (for simplicity, we assume only bi-directional edges, i.e., edge $e_{i,j}$ is equivalent to edge $e_{j,i}$, but are oppositely directed).

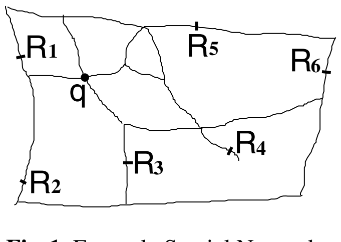


Fig. 1. Example Spatial Network

Two sets of *points*, termed *query points* and *data points* are also assumed. Each such point p is either a vertex or a two-tuple $s = (e_{i,j}, pos)$ consisting of an edge $e_{i,j}$ and a travel length pos from v_i along edge $e_{i,j}$.

Next, the graph and data points must be stored on disk using an appropriate data structure (e.g., the CCAM structure [14]). Then a query such as the example query from above is processed by accessing this data structure.

The example spatial network and data points in Figure 1 are represented as shown in Figure 2. To find the nearest neighbor among $dp_1, \ldots, dp_6$ of the query point, which is located at vertex v_6, a graph search is performed. For example, an algorithm similar to Dijkstra's algorithm or the A* algorithm [11] can be used to incrementally search the graph starting at v_6 until the nearest neighbor is found.

To be more specific, the INE search algorithm [12] uses two priority queues, Q_v for adjacent vertices and Q_{dp} for data points. Given the query point $q = v_6$, adjacent vertices of q are visited and out into Q_v (i.e., $Q_v = \langle(v_7, 1), (v_5, 2), (v_9, 3), (v_2, 3)\rangle$). As no data points are found, the vertex v_7 in Q_v is dequeued, and its adjacent vertices v_3 and v_8 are inserted, i.e., $Q_v = \langle(v_5, 2), (v_3, 2), (v_8, 2), (v_9, 3), (v_2, 3)\rangle$. Data point dp_1 is found when v_5 is dequeued and its adjacent vertices are accessed ($Q_{dp} = \langle(dp_1, 3)\rangle$). The process continues until the minimum distance from q to a vertex in Q_v is no smaller than the distance to the nearest data point (i.e., 3).

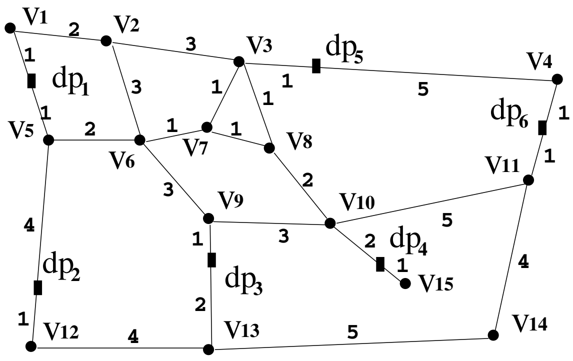


Fig. 2. Example Network

This process, termed *incremental network expansion*, has been used as an ingredient in most existing algorithms for spatial-network queries, including *(continuous) k nearest neighbors* ((C)KNN), *reverse nearest neighbors* (RNN), *k closest pairs*, *e-distance join*, and *aggregate nearest neighbors* (ANN).

Nearest neighbor queries have been studied extensively in settings where the Euclidean distance is assumed and R-trees are used. The problem of computing KNN queries in spatial network databases has only been addressed more recently. Early work in this direction presented a general framework for supporting NN queries, that included a detailed data model and hierarchical search algorithms [8]. Subsequent work has considered the optimization of the disk accesses to the network data and data points during query processing.

The INE algorithm and extensions of it have been used for computing range queries, KNN queries, k closest pair queries, and e-distance joins [12]. This approach works well for dense data points, but it entails excessive accesses to the network data when the data points are sparse. To improve performance, pre-computation techniques have been proposed for primarily KNN and CKNN queries. In the next section, we review three such proposals, namely the VN3 [9], Islands [3], and SPIE [5] approaches.

While these approaches represent significant advances, they also have shortcomings. The data points are assumed to be known to the system prior to any queries. While this assumption will work in some (important) settings, it does not make for a versatile solution. Another limitation is that all data points are assumed to be relevant for all queries, while each query will generally concern only certain types of data points. Next, the approaches do not contend well with updates. Queries have to be "frozen" until the updates (including costly re-pre-computations) have been performed. The query performance then depends on the performance of the updates.

Motivated by these limitations and the need for *versatile* techniques, we propose a novel and more general pre-computation structure, the S-GRID (Scalable Grid), that enables the efficient computation of a broad range of query types in spatial networks. Results of experimental studies show that the S-GRID provides excellent performance in comparison to its competitors.

In summary, the S-GRID approach optimizes the network expansion inherent to many query processing algorithms in spatial networks, while offering the following features.

1. Unlike the existing pre-computation approaches, the S-GRID does pre-computation solely on the network data, and data points are associated with the pre-computed data only at query time.
2. With the existing approaches, updates may cause queries to pause until the updates are complete. With the S-GRID, queries affected by an update are able to expand on the original network data inside the grid cell being updated.
3. While the handling of traffic restrictions such as one-way streets and turn restrictions at intersections is difficult with the existing approaches, the S-GRID encapsulates these into a simple, virtual network so that the query processing can be kept simple and efficient.

The rest of the paper is organized as follows. Section 2 explores previous techniques for efficient KNN query processing in spatial networks. The next section presents the details of the solution. Section 4 empirically compares this solution to existing algorithms. Finally, Section 5 summarizes the paper and suggests research directions.

2 Related Work

As mentioned already, network expansion has been used in a range of query processing algorithms, including algorithms for KNN [12,4], CKNN [1], RNN [18], and ANN [17] queries, as well as for data clustering [16]. We proceed to consider briefly two early proposals for KNN query processing and then proceed to consider three additional proposals in some detail.

The first proposal is to transform a road network to a high-dimensional Euclidean space in which the traditional KNN search algorithms can be applied [13]. This transformation involves the off-line pre-computation of the network distances between all pairs of vertices, and it uses high-dimensional spatial indexes. As a result, the proposal is of limited interest in our setting.

The next proposal involves two approaches to KNN query processing, namely the INE approach already explained and an approach called IER that exploits Euclidean distances for achieving better performance [12]. To find the KNNs of a query point q, this approach uses an incremental KNN algorithm to find the nearest neighbors in Euclidean distance. These are then sorted in ascending order of their network distance to q and the distance of the k-th neighbor is denoted as d_{max}. These KNNs and d_{max} are being maintained while subsequent Euclidean neighbors are retrieved incrementally, until the next Euclidean nearest neighbor has larger Euclidean distance than d_{max}. It is shown that the INE approach clearly outperforms the IER approach. Further, the IER approach is inapplicable for all notions of distance such as travel time.

Inspired by the use of Voronoi diagrams in Euclidean spaces, the Voronoi-based Network Nearest Neighbor approach (VN3 [9]) generates a Network Voronoi Diagram based on a given set of data points and pre-computes the network distances within each generated Voronoi polygon. Nearest neighbor computations can then utilize the pre-computed distances.

Figure 3 shows the network Voronoi diagram for our example. The Voronoi polygon of data point dp_3 contains four border points: b_1, b_2, b_3, b_4. The first nearest neighbor of query point q can be directly found as dp_3 because q is inside the Voronoi polygon of dp_3. To find the next nearest neighbors, the data points of the neighboring Voronoi polygons are collected as the candidate set. Then a refinement is applied to find the actual network distance from q to these data points. Specifically, a network expansion is made to find the network distances from q to the border points of dp_3. Then the distance from q to the neighboring data points can be found by adding the query-to-border distances to the pre-computed border-to-data point distances of the adjacent polygons. As shown in Figure 3, since the distance from q to b_2 is 2.5 and the pre-computed distance from b_2 to dp_4 is 2.5, the network distance from q to dp_4 is 5.

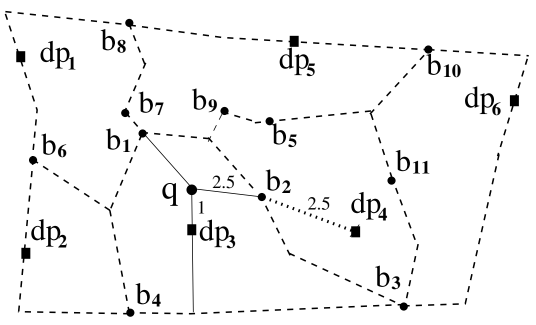


Fig. 3. NN Search with VN3

The VN3 approach excels when there are few data points. One limitation is that a Voronoi diagram cannot be generated without knowledge of all the data points. Another is that it does not contend well with queries that concern only data points of certain types. Thus, the approach of generating a Voronoi diagram for each type of data points, such as Thai restaurants, museums, and shopping malls, renders it difficult to process "multi-type" queries such as "find the k nearest Thai and Chinese restaurants and museums" (or "sub-type" queries such as "find the k nearest modern art museums"). A service that identifies all nearest friends will need a diagram for each unique set of friends.

The Islands approach provides versatility as it enables to control the amount of pre-computation done in preparation for computing KNN queries [3]. First "islands" are pre-computed: starting from each data point, all vertices that are within a given radius r_{min} to the data point are part of the data point's island, and the distance to the data point for each such data point is recorded.

A KNN query processing algorithm then makes network expansions from the query point while using the pre-computed "islands" encountered during the expansions.

Using a radius of 3, Figure 4 shows the islands generated for our example. Here, query point q is inside the island of dp_3, which is the nearest neighbor of q. To find subsequent nearest neighbors, a network expansion is made from q. Since vertex v_{10} is inside the island of dp_4 and v_6 is inside the islands of dp_1 and dp_5, we can find their network distance to q as $D_N(q, dp_4) = 6, D_N(q, dp_1) = 7, D_N(q, dp_5) = 7$. The expansion continues until the distance from q to the next vertex plus r_{min} is no smaller than the distance to the kth nearest neighbor.

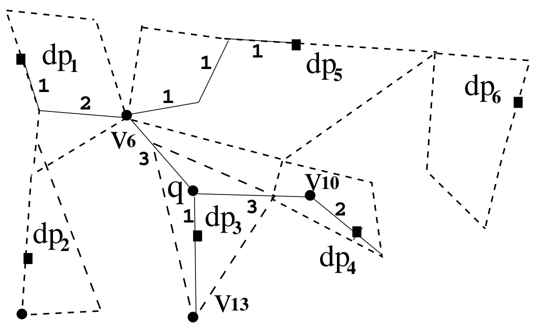


Fig. 4. NN Search with Islands

In the Islands approach, it is possible to control the sizes of the islands (i.e., the radius value r_{min}). This offers flexibility in balancing the amount of pre-computation data, the cost of updating the pre-computed data, and the efficiency of the KNN queries. But as for the VN3, the Islands approach requires a pre-knowledge of all the data points. And when updating the pre-computation data in both approaches, all the KNN queries "covering" the network area that is being updated must wait for the update to complete. Thus, the update performance affects the query performance.

The SPIE approach reduces the network into a set of inter-connected *shortest path trees* (SPTs) [5]. Dijkstra's algorithm is adapted to grow the SPTs, which are later transformed into SPIEs (Shortest Path tree with horizontal edges and triangular Inequality Edges). Making the assumption that the data points are located on the network vertices, an index is built that stores, for each tree node, the nearest data points in its descendants as well as the distances. The KNN queries are then processed on the SPIEs instead of the real network.

The assumption that the data points are located on network vertices is a limitation in many situations. In practice, data points are located on the edges and are represented using linear referencing [2]. Adding a vertex for each data point on an edge will substantially increase the size of the network. Next, transportation networks often involve one-way streets, u-turn restrictions, and turn-restrictions at intersections. As a result, although the SPIE approach yields a nice reduction of a network that is a simple undirected graph, the same reduction process does not apply when the constraints and restrictions on edges and vertices are considered.

Previous work on hierarchical structures for path-finding in road networks [7] is not closely related to this paper's contribution. While that work focuses on shortest-path computations, this paper focuses on KNN and other queries. Also, the S-GRID does not use a hierarchical structure, but a simple 2D grid to organize the pre-computation process and direct the KNN queries.

Summarizing the existing solutions, three basic techniques exist that aim to reduce the cost of expensive network expansions. First, by pre-computing network distances between the vertices of the network and the data points in a specific area of the network, the network expansion covering this area can be avoided by instead looking up the distance values. For example, in the VN3 approach, expansions in each Voronoi cell can be avoided as there is only one data point in each cell and the pre-computed border-to-border and border-to-data distances are used. Second, by linking more data points with each vertex, the query algorithm has knowledge of more candidate nearest neighbors at earlier stages of a network expansion, which restricts the expansion scope. For example, in the Islands approach, the k nearest neighbors can be found immediately if the query point is inside at least k islands. Third, by simplifying the network (as in the SPIE approach), queries are processed more efficiently because the expansions are applied on a sparser network.

Common to all of the above approaches is that the pre-computations are data-point dependent: distances to data points are pre-computed (in all three approaches) and the network is subdivided based on the positions of the data points (in the VN3 approach). However, as mentioned in the introduction, this dependence on the data points is often either undesirable or not possible at all. Thus, in contrast to the previous research, we make the fundamental assumption that the data points and their positions are known to the system only at query time. This yields a much more versatile solution. The challenge then becomes one of achieving competitive performance while using only data-point independent pre-computations.

The idea of network partitioning and network connectivity indexing is extensible to other application domains where graphs are queried. For instance, a recent paper [6] introduces a similar divide-and-conquer approach for keyword searches on graphs. The proposed BLINKS approach uses heuristic-based algorithms to partition the graph. In contrast, the S-GRID takes advantage of the spatial embedding of the network and employs a regular spatial grid to partition the network.

3 The S-GRID Approach

The S-GRID approach is so named because it employs a 2-dimensional grid for "summarizing" a network and performing pre-computations. In particular, distance pre-computations are made that involve the intersection points between the grid and the network. These grid-based pre-computations usually simplify the network—the grid-network intersection points together with the connections among these points form a simpler, virtual network. At query time, it is possible to link, in a simple and efficient way, queries and data points to the pre-computations.

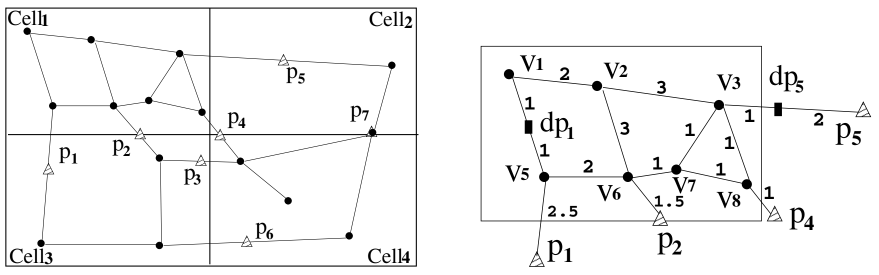
By varying the number of cells, the trade-offs between query performance, update performance, the pre-computation costs, and the size of the pre-computation data can be tuned. Although grids have been used for a variety of purposes in many contexts in spatial databases (e.g., in [15]), we believe that our use of a grid in the context of a spatial network database is novel. We proceed with the details of the new approach.

3.1 Grid Partitioning and Pre-computation

As described, we apply a 2-dimensional grid to the spatial network. The part of the network inside a grid cell forms one or more mutually disconnected sub-networks. A vertex belongs to a grid cell if its coordinates place it inside the cell. If the two vertices of an edge belong to different cells, we define the midpoint of this edge as a *border point* between the two cells. A vertex with coordinates that intersect with a grid boundary is also a border point. For example, points $p_1, p_2, \ldots, p_7$ in Figure 5(a) are the border points when we apply the shown 2×2 grid to the network in Figure 2 (note that p_7 is a vertex while the others are midpoints). We model each grid cell as a three-tuple $ce = (V, BP, DP)$ where V is the set of vertices belonging to the cell, BP is the set of border points of the cell, and DP is the set of data points inside the cell. Next, if a vertex or a border point is connected to another border point through a sub-network of the cell, the length of the shortest path connecting them in the sub-network is termed their *connected distance* in the cell. For instance, in Figure 5(b), the connected distance between vertex v_2 and border point p_2 in Cell_1 is 4.5.

For each cell, we pre-compute two types of distance values: (a) the connected distance for each pair of connected border points; (b) the connected distance for each pair of a connected vertex and a border point. The spatial network as well as the pre-computation data are stored on disk in the *Vertex-Edge*, *Cell-Border*, and *Vertex-Border* components. The *Vertex-Edge* component corresponds to a similar structure in the INE and Islands approaches [3,12]: adjacency lists of vertices are mapped to disk pages based on the Hilbert values of the vertices.

For our example, page pg_1 in Figure 6 contains the adjacency lists of vertices v_1 and v_2. Each entry in an adjacency list corresponds to an edge in the graph and contains the



(a) A 2×2 Grid (b) Cell_1 in Detail

Fig. 5. Example Grid Partition

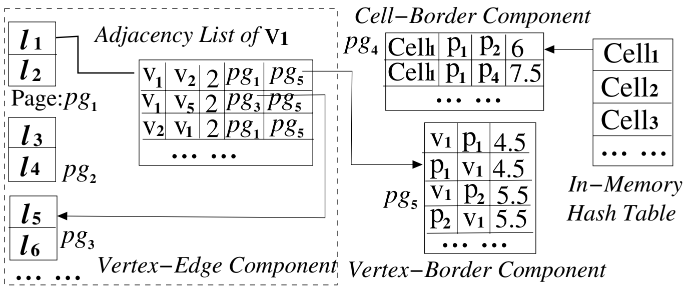


Fig. 6. Example Data Structure

identifications of the start vertex (e.g., v_1) and the end vertex (e.g., v_5), the length of the edge ($e_{1,5} = 2$), a pointer to the disk page (pg_3) containing the adjacency list of the end vertex (v_5), and a pointer to the disk page (pg_5) in the *Vertex-Border* component that stores the connected distances between the start vertex (v_1) and the border points of the cell to which this vertex belongs (Cell_1).

The *Cell-Border* component stores the distances between the border points of the grid cells. For example, in Figure 6, the entries in page pg_4 record the connected distances between border points p_1, p_2, p_4, p_5 of Cell_1 in Figure 5(b). The in-memory hash table links each grid cell to its corresponding disk pages in the *Cell-Border* component. Note that no data points are represented in these data structures.

The computation of connected distances among the border points and vertices in a cell is simple. From each border point, a network expansion is made in the sub-network of the cell following the edges in the reverse direction to find the distances from the reachable vertices. For each vertex discovered in the expansion, the distance to the border point is recorded in the *Vertex-Border* component. The expansion stops when an edge contains another border point; in this case, the connected distance between the two border points is recorded in the *Cell-Border* component.

The resulting pre-computation data structure has a number of features useful when processing queries and performing network updates.

First, with the 2-dimensional grid and the *Vertex-Border* component, it is easy to link the data points with the pre-computation data. We introduce a function **discoverDataPoints**(DP, ce) that returns all the data points from the set DP that belong to the grid cell ce. The function scans the data points in DP. A data point $dp = (e, pos)$ belongs to a cell if its nearest vertex on edge e belongs to this cell. The cell memberships of data points can also be maintained dynamically as data points are inserted, deleted, or updated. Finally, for each of the returned data points dp, the function returns a set of entries $(p, dp, d(p, dp))$, where p is a border point and $d(p, dp)$ is the connected distance from p to dp in ce. It is straightforward to compute these distances, since distances from end-vertices of edge e to border points of the cell can be found in the *Vertex-Border* component.

Second, the border points together with the links among them form a *virtual network*. Unless the grid is very dense, the shortest path connecting two border points in the virtual network has fewer edges than the shortest path connecting the same points in the underlying network. Note that if no pre-computation is done, to find a data point that is one of the nearest neighbors of a query point, the shortest path between the query point and this data point has to be traversed in the underlying network. If this shortest path traverses border points $p_1, \ldots, p_b$ (where $b \geq 2$), a sub-path connecting border points p_i and p_{i+1} is a shortest path between these two points in some cell of the grid, i.e., it corresponds to an edge in the virtual network. Thus, once a network expansion reaches a border point, it can proceed more efficiently in the virtual network. The above-described **discoverDataPoints** function extends the virtual network with data points so that they can be discovered via expansion in the virtual network.

Third, update operations to the pre-computation data are local and data-point independent. Specifically, when a vertex or an edge is added, modified, or deleted from the *Vertex-Edge* component, network expansions from each of the border points of the corresponding grid cell should be made to refresh the distances. Since we only compute the connected distance inside one cell, the cost of update operations is limited to the sub-networks inside this cell. Thus, by varying the number and thus size of the grid cells, the cost of updates can be controlled. In addition, when the network expansions of queries visit cells that are being updated, these expansions can use the underlying network instead of the virtual network in these cells. This way, network update operations do not block querying.

Finally, as mentioned, the number of grid cells can be varied. If the data point density is low, a sparse grid (i.e., with large cells) improves query performance as the expansion process is on a virtual network with fewer vertices. In contrast, a dense grid will have better update performance since the part of the network influenced by an update becomes smaller. In the two extremes, i.e., when there is only one cell in the grid or the grid cells become too small, the approach is not efficient. However, by tuning the cell size, it is possible to achieve improved update and query performance.

We proceed to describe how S-GRID is used to process the KNN query.

3.2 KNN Query Processing

We adapt the INE algorithm [12] to compute the KNN query using the S-GRID. Briefly, given a query point q, a value k, and a set of data points DP, we first start a network expansion, termed the *inner expansion*, in the cell where q is located. Whenever a border point is reached, the *outer expansion* proceeds from that point. The outer expansion is an expansion on the virtual network formed by the border points and their links.

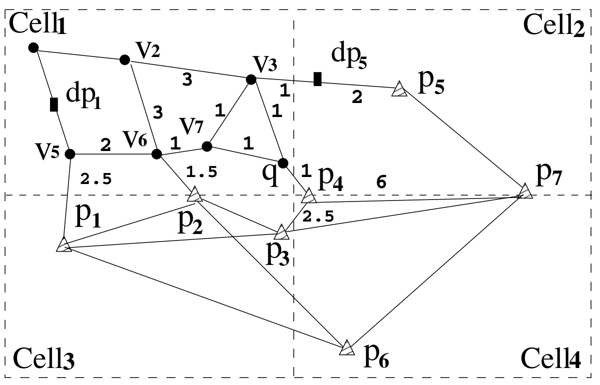
If the cell holds no data points or when the shortest paths to all data points inside the cell have been discovered, the inner expansion is stopped. The *Vertex-Border* component is used to traverse directly from inside a cell and into the virtual network. When the outer expansion visits a border point, the **discoverDataPoints** function is used to find all data points in the cells that share this border point. This process continues until k nearest neighbors are found.

We provide the pseudo code of the KNN algorithm in the following. In addition to the three above-mentioned parameters, the algorithm gets a set of grid cells CE as a parameter. We use two priority queues, Q_{dp} and Q_v, to record, respectively, data points and vertices (or border points) together with their distance to the query point, denoted as $d(q, dp)$ and $d(q, v)$. Both queues sort elements by the distance value and do not allow duplicate data points or vertices. The size of Q_{dp} is limited to k elements.

Both queues have *update* and *deque* operations. The *update*(dp/v, *dist*) operation inserts a new data point or vertex and the corresponding distance into the queue. If this data point or vertex is already in the queue then, if *dist* is smaller than the distance stored in the queue, the distance value in the queue is updated to *dist*. The *deque* operation removes a vertex or a border point with the smallest distance and returns it. Another in-memory list L caches all the "discovered" data points returned by the **discoverDataPoints** function. Queues Q_v and Q_{dp} and list L are assumed to be empty initially.

```
(1)  procedure $KNN(q, k, DP, CE)$
(2)  $sort(DP, CE)$ // put $DP$ into subsets based on cells
(3)  Let the subsets of $DP$ be $ce_1.DP, \ldots, ce_m.DP$
(4)  $ce_q \leftarrow findcell(q)$
(5)  for each $dp \in findDP(q.e, ce_q)$: $Q_{dp}.update(dp, d(q, dp))$
(6)  for each $v \in \{q.e.v_s, q.e.v_e\}$ $Q_v.update(v, d(q, v))$
(7)  for each $bp \in ce_q.BP$: $Q_v.update(bp, d(q, bp))$    // Vertex-Border is used
(8)  $Q_{dp} = \langle(dp_1, d(q, dp_1)), \ldots, (dp_k, d(q, dp_k))\rangle$
(9)  $d_k \leftarrow d(q, dp_k)$ // $d_k \leftarrow \infty$ if $dp_k = \varnothing$
(10) $v_x \leftarrow Q_v.deque$, mark $v_x$ as visited
(11) while $d(q, v_x) < d_k \wedge Q_v \neq \emptyset$
(12)   if $v_x$ is a vertex
(13)     for each non-visited adjacent vertex $v_y$ of $v_x$
(14)       for each $dp \in findDP(e_{x,y}, ce_q)$
(15)         $Q_{dp}.update(dp, d(q, v_x) + d(v_x, dp))$
(16)       if $v_y \in ce_q.V$: $Q_v.update(v_y, d(q, v_x) + e_{x,y}.l)$
(17)     if $ce_q.DP = \emptyset \vee (ce_q.DP \subset Q_{dp} \wedge \forall dp \in ce_q.DP, d(q, dp) \leq d(q, v_x))$
         // shortest paths found to all $dp \in ce_q.DP$
(18)       prune each $(v, d(q, v))$ from $Q_v$ if $v$ is a vertex
(19)   else // $v_x$ is a border point; switch to the virtual network
(20)     for each $ce_{ki} \in findcells(v_x, CE)$
(21)       if $ce_{ki} \neq ce_q \wedge |ce_{ki}.DP| > 0 \wedge ce_{ki}$ is undiscovered
(22)         $L \leftarrow L \cup$ discoverDataPoints$(DP, ce_{ki})$
(23)         mark $ce_{ki}$ as discovered
(24)       for each non-visited adjacent border point $v_y \in ce_{ki}$ of $v_x$
(25)         $Q_v.update(v_y, d(q, v_x) + d(v_x, v_y))$    // Cell-Border is used
(26)     for each $(v_x, dp_{xi}, d(v_x, dp_{xi})) \in L$
(27)       $Q_{dp}.update(dp_{xi}, d(q, v_x) + d(v_x, dp_{xi}))$
(28)   $d_k \leftarrow d(q, dp_k)$
(29)   $v_x \leftarrow Q_v.deque$, mark $v_x$ as visited
(30) return $Q_{dp}$
```

The algorithm first partitions the data points in DP according to the cells they belong to. Then the network expansion begins with the part of the network inside the cell ce_q (lines 12–16). The border points of ce_q are treated as additional vertices of the network. When the shortest paths to all the data points in ce_q have been computed, the inner expansion on the actual network is completed (lines 17–18), and the algorithm continues only with the outer expansion on the virtual network (lines 19–27). When border points are visited by the outer expansion, the algorithm discovers data points in



(a) 2NN at q

Step	Q_v	Q_{dp}	d_k
1	$(v_7, 1), (v_3, 1), (p_4, 1), (p_2, 3.5),$ $(p_5, 4), (p_1, 6.5)$	$\emptyset$	∞
2	$(v_3, 1), (p_4, 1), (v_6, 2), (p_2, 3.5),$ $(p_5, 4), (p_1, 6.5)$	$\emptyset$	∞
3	$(p_4, 1), (v_6, 2), (p_2, 3.5), (p_5, 4),$ $(v_2, 4), (p_1, 6.5)$	$(dp_5, 2)$	∞
4	$(v_6, 2), (p_2, 3.5), (p_3, 3.5), (p_5, 4), \ldots$	$(dp_5, 2), (dp_4, 4)$	4
5	$(p_2, 3.5), (p_3, 3.5), (p_5, 4), \ldots$	$(dp_5, 2), (dp_4, 4)$	4
$6, 7$	$(p_5, 4), \ldots$	$(dp_5, 2), (dp_4, 4)$	4

(b) Running Steps

Fig. 7. Example KNN Query

the cells that sharing the border points (lines 20–23). Note that it is possible for inner and outer expansions to run concurrently, which may happen, for example, if the query point is quite close to the border of a cell. The algorithm guarantees that both expansions will stop if KNNs are found.

The algorithm uses three auxiliary functions. Function $\mathit{findDP}(e_{x,y}, ce)$ returns the data points in $ce.DP$ that are located on one of the edges $e_{x,y}$ and $e_{y,x}$ and also belong to cell ce. Function $\mathit{findcell}(q)$ returns the cell that q belongs to, i.e., the cell that its nearest vertex, either $q.e.v_s$ or $q.e.v_e$, belongs to. Finally, $\mathit{findcells}(p, CE)$ returns the cells that have p as a border point.

To illustrate the working of the algorithm, consider a 2NN query at vertex v_8 in Figure 2. The algorithm first scans the network inside the cell of query point $q = v_8$— see Figure 7(a). The border points of the cell are also inserted into Q_v based on their distance to q. The steps in computing the query are listed in Figure 7(b). In step 3, since the border point p_4 is closer to q than other vertices such as v_6, the algorithm discovers data points in cell$_4$, and data point dp_4 is found through p_4. After data points dp_5 and dp_4 are found, steps 6 and 7 continue and visit adjacent points (in the virtual network) of p_2 and p_3, and then the algorithm stops.

The KNN algorithm with the S-GRID improves the efficiency of the INE algorithm in three ways. First, the inner expansion is avoided fully or in part, if there are no data points in the cell of the query point or if all these data points have been reached, respectively. Second, by doing the inner and outer expansions concurrently, more data points are inserted into Q_{dp} early in the algorithm, which restricts the expansion scope. Finally, the expansion on the virtual network formed by the border points and their links utilizes the pre-computation data to link data points with border points, which makes it possible to find these data points in the virtual network.

In some cases, these optimizations will not take effect. For instance, when there are more than k data points in the cell of the query point and the query point is close to the center of the cell, all the nearest neighbors needed may be found by the inner expansion, in which case the efficiency of the S-GRID algorithm is equal to that of the INE algorithm. To improve this, the system can maintain several S-GRIDs with different cell sizes and assign a proper S-GRID to run the KNN algorithm based on the location of the query point and the data point density.

3.3 Extensions

The S-GRID approach is useful for the computation of many different kinds of queries. To illustrate this, we describe how the S-GRID can be used for computing range and CKNN queries.

Range Query. The *range* query retrieves all data points that are within a given network distance R of a query point. Intuitively, the same network expansion process can be used for the range query as for the KNN query, except that the termination condition in line 11 of the *KNN* algorithm has to be changed to $d(q, v_x) < R \wedge Q_v \neq \emptyset$. Note that the S-GRID is used in the same way to maintain the inner and the outer expansions.

CKNN Query. The *CKNN* query retrieves k nearest data points along a given query path, i.e., it finds k nearest neighbors to any point of a given path in the network. Existing solutions for CKNN query in spatial networks [1,10] depend on an efficient algorithm for the static KNN query. Specifically, as indicated by the most recent proposal, UNICONS (UNIque Continuous Search) [1], to perform a CKNN query on a path $n_i, n_{i+1}, \ldots, n_j$, it is sufficient to retrieve data points directly on the path and then run a static KNN query at each vertex n_k on the path $(i \leq k \leq j)$ [1, Lemma 2]. To improve the efficiency of such KNN queries, the UNICONS approach pre-computes and stores KNNs of a selected nodes in the network. The S-GRID approach can be used for processing the static KNN queries at the path nodes. Similar to the UNICONS approach, we can also store KNNs of every border point of the S-GRID so that when a KNN query reaches a border point, it can re-use the KNNs of this border point and does not need to expand further from this point.

Accommodating Traffic Regulations. The S-GRID approach only requires few modifications in order to be able to contend with traffic regulations when computing KNN queries. Specifically, when one-way roads, streets with u-turn restrictions, and road junctions with turn restrictions have to be considered in the expansion process, only the inner expansion needs to check these constraints. The outer expansion on the virtual network needs not contend with such restrictions, as they have already been addressed during pre-computation.

4 Empirical Evaluation

4.1 Settings

To gain insight into the performance of the S-GRID, we conduct experiments with two datasets. The first represents the real-world road network and points of interest in Aalborg (AAL), Denmark, and it contains $11,300$ vertices, $13,375$ bi-directional edges, and 279 data points. The second dataset is the road network data of San Francisco (SF) (downloaded from http://www.fh-oow.de/institute/iapg/personen/brinkhoff/generator/), which contains $175,343$ vertices and $223,140$ bi-directional edges. For the SF dataset, we use synthetic data points that are generated randomly with a density of 0.1% (the density is the number of data points versus the number of bi-directional edges in the network).

The road network and pre-computation data are arranged into disk pages based on the data structures described in Section 3.1. We set the page size to 4k and use an LRU buffer for caching the disk pages read by the algorithms. The total size of the LRU buffer is 15% of the network data (i.e., the *Vertex-Edge* component). The AAL and SF datasets contain, respectively, 129 and $4,023$ pages in the *Vertex-Edge* component.

We compare with the INE and the Islands approaches [3], and the consider the performance of these approaches in terms of the CPU running time and the amount of disk I/O operations. All the tested approaches are implemented in C++ and performed on a Pentium IV 1.3 GHZ processor with 512 MB of main memory and running Windows 2000. Query points are randomly generated in all the experiments. Each reported performance number is the average number obtained after measuring the performance in several runs of the experiment.

Two series of experiments are conducted. The first series studies the performance of the KNN query comparing the S-GRID, the INE, and the Islands approaches. In these experiments, we vary k, the density of the data points, and the number (and size) of grid cells. The second series of experiments examines the cost of pre-computation and update operations in the Islands and S-GRID approaches.

4.2 Experiments on KNN Query Performance

In this experiment, we examine how the performance of KNN queries is related to the value of k, the density of data points, the size of the grid cells in the S-GRID approach, and the island size in the Islands approach. We set $k = 5$ for the experiments with the density, the grid cell size, and the island size. The AAL and SF networks use grids of size 8×8 and 20×20, respectively.

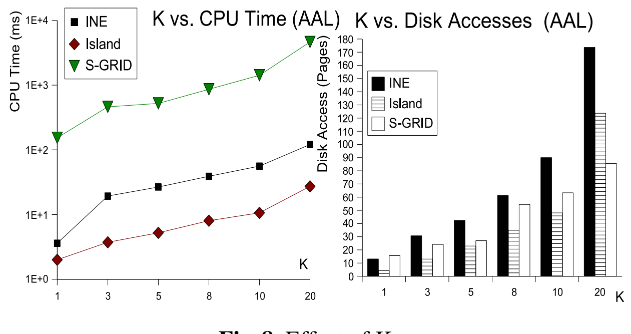


Fig. 8. Effect of K

To express the island size, we define the maximum Euclidean distance between all vertices in the road network as D_{max}. The island radius used is then represented as a fraction of D_{max}. In all experiments, the islands of the same network have the same radius. In the case where the island size is less than the edge length, we set the island to cover the edge of the data point. The AAL network uses an island size of $0.05D_{max}$ while the SF network uses $0.01D_{max}$.

As shown in Figures 8 and 9, the *KNN* algorithm with the S-GRID requires more CPU time than the INE and Islands approaches. This is because each vertex in the virtual network has more adjacent edges than the original spatial network. This increases the insertion and sorting times in queue Q_v of the network expansion algorithm. The S-GRID requires fewer disk accesses than INE, but is slightly worse than the Islands approach. The superior performance of the Islands approach (when compared to the S-GRID) is due to the usage of pre-computed distances to the queried data-points in the Islands approach. The slightly lower performance of the S-GRID is the price that is paid for the flexibility of not having to know the set of data points at the time of pre-computation.

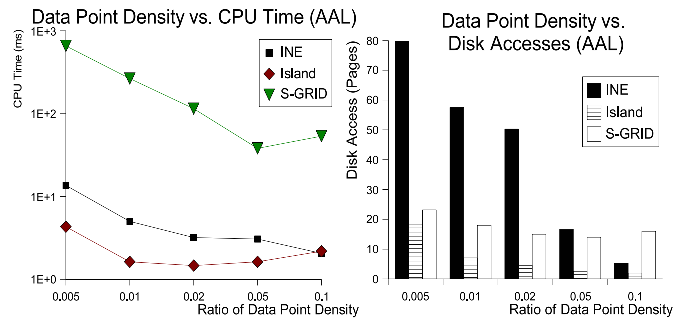


Fig. 9. Effect of Data Point Density

Figure 10 reports on experiments where the S-GRID cell size is varied. As expected, the results of the experiments show that when the number of cells increases, the performance of the KNN queries improves. However, as illustrated in Figure 10, when the cells get too small, there are too many border points and links, which increase the cost of expansions on the virtual network to the point where it becomes even more expensive than just making expansions on the original network.

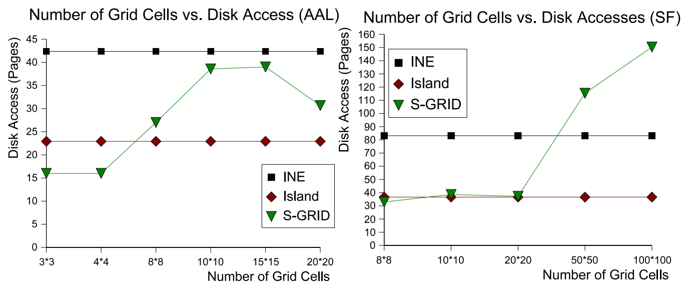


Fig. 10. Effect of Number of Grid Cells

4.3 Experiments with Pre-computation and Update

With the objective of exploring the cost of pre-computations with the Islands and S-GRID approaches, the second series of experiments measure the disk I/O and number of generated data items (i.e., the amount of border points, links, and pairs of distances)

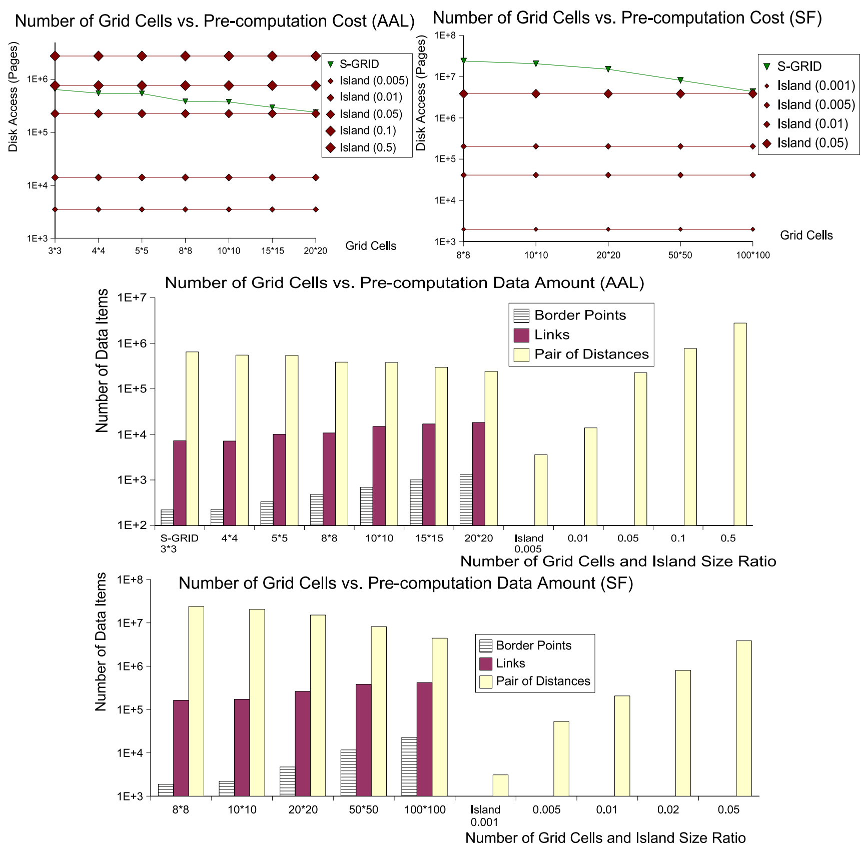


Fig. 11. Pre-computation and Storage Costs—S-GRID and Islands

during pre-computation. We do not report the time for writing the pre-computation data to disk as this time is proportional to the number of generated items. Updates to the network edges and vertices are relatively rare when only the spatial coordinates and the topology of the network are considered. However, for applications where the distance in the road network is measured as the travel time, the frequency of network updates can surpass even the frequency of queries. That is why we also study the network update performance of the compared pre-computation approaches.

The "pair of distances" recorded in Figure 11 shows the number of distances (i.e., the distances from vertices to data points or border points) collected during pre-computation. The figure shows that the pre-computation cost for the S-GRID is higher than the corresponding cost for the Islands approach for small islands, but that it decreases with increasing numbers of grid cells since the network expansion scope from each border point gets smaller.

To illustrate the difference between the original network and the virtual network of the S-GRID, we list the number of vertices and edges (edges $e_{i,j}, e_{j,i}$ are counted as one) in the original AAL and SF networks as well as the corresponding virtual networks.

As shown in Figure 12, the 3×3 and 8×8 grids on the AAL network have fewer vertices and edges, which can lead to improved KNN query performance (as in Figure 10). Note that the 15×15 grid on the AAL network produces fewer vertices, but more edges, when compared to the original network. Nevertheless, an order of magnitude reduction in the number of vertices (for the 15×15 grid) results in improvements of the query performance (compared to the INE in Figure 10). Similar to this, the 10×10 and 20×20 grids on the SF network improve the query performance. When the grid is too dense, i.e., 50×50 or 100×100, there are too many edges in the virtual network, which negatively effects the efficiency of the query algorithms. The space consumption of the S-GRID, while dependent on the number of grid cells, is generally larger than the space consumption of the Islands approach (see Figure 11). Again, the space is sacrificed for the flexibility of the S-GRID. To discover the appropriate number of grid cells for the specific network and data sets, an iterative approach can be used which, by running a certain amount of test queries over several different grid partitionings, chooses the number of grid cells that results in the most efficient execution of the test queries.

To examine the cost of updates in the S-GRID and the Island approaches, we randomly pick one edge in the AAL network and vary its length so as to collect the CPU and disk I/O costs of the re-computations of pre-computed data. We vary the amount of grid cells and the size of islands and measure the update cost.

As illustrated in Figure 13, the update cost in the S-GRID decreases as the cells get smaller since the cost of doing network expansions is smaller for smaller cells. The cost of doing updates on the islands increases dramatically as the islands increase in size. With small islands, it is very likely that an edge is not in any island, in which case the update cost is close to zero (one only needs to update the network data). When the island size increases, each edge is likely associated with more than one island so that the update operation has to re-generate more islands.

Network	Vertices	Edges
AAL	11, 300	13, 375
AAL (3 × 3 Grid)	221	7, 282
AAL (8 × 8 Grid)	485	10, 774
AAL (15 × 15 Grid)	1, 006	16, 986
SF	175, 343	223, 140
SF (10 × 10 Grid)	2, 213	172, 275
SF (20 × 20 Grid)	4, 726	262, 187
SF (50 × 50 Grid)	11, 692	381, 705
SF (100 × 100 Grid)	22, 822	419, 307

Fig. 12. Reduction of Network Size

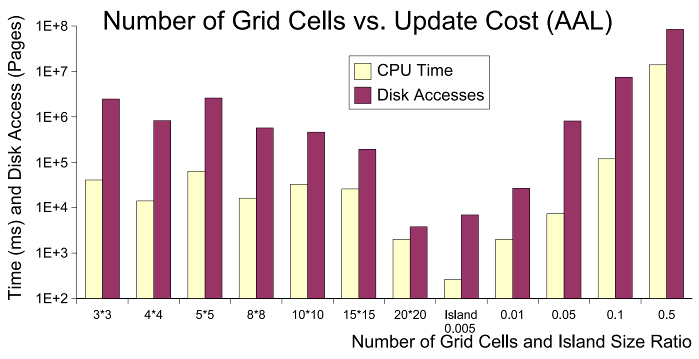


Fig. 13. Update Cost of S-GRID

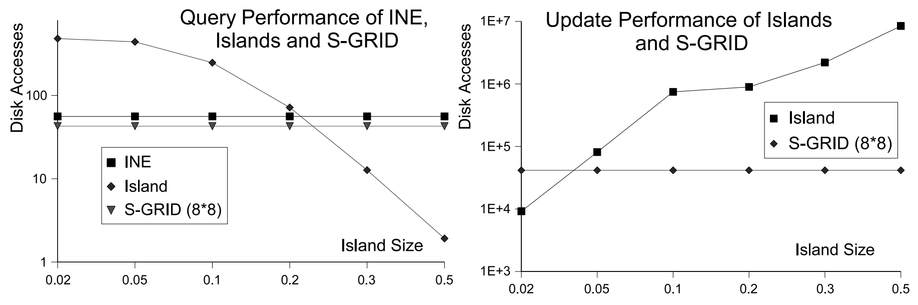


Fig. 14. "Sub-Category" Query and Update

To test how efficiently the S-GRID and other approaches perform "sub-type" KNN queries that search for NNs belonging to a sub-type of data points in the dataset, we randomly divide the 279 data points in AAL network into 10 groups (with 28 or 27 data points in each group) and implement KNN query algorithms that use the INE, S-GRID, and Islands approaches for finding KNNs in one of the groups. The default value of k is 5 and we use an 8×8 grid. We vary the size of the islands.

To support sub-type queries, the Islands approach creates islands for all the data points and the online expansion algorithm checks if a newly-discovered island belongs to the target category, to determine whether further expansion is necessary. In addition, we show the update cost (changing one edge weight) of the Islands and S-GRID approaches. As demonstrated in Figure 14, for the KNN query, the S-GRID approach requires fewer disk accesses than the INE, and the island size has to grow to $0.3D_{max}$ for the Islands approach to have better performance than the INE and S-GRID. However, the cost of updates in the Islands approach with islands of even smaller size (e.g., $0.05, 0.1, 0.2$ of D_{max}) is worse than the update cost of S-GRID. Thus, in terms of overall performance of processing sub-type queries and processing updates, the S-GRID is better than the Islands approach. Note that, in the reported experiments, all data points are divided only into 10 "sub-types." In real applications, the data points may be divided into much more "sub-types," which further increases the advantage of the S-GRID over the Islands approach.

5 Summary and Future Work

Spatial network databases have gained substantial attention with the development of advanced positioning and mobile communication and computing technologies. One current focus is on how to reduce the amount of disk accesses needed for executing spatial and spatio-temporal queries in spatial network databases.

In particular, different approaches to pre-computation has been studied with the purpose of achieving efficient query processing. Motivated by the limitations of existing pre-computation approaches, this paper proposes a more versatile approach, termed the S-GRID, to pre-computation.

In a world where few query processing and indexing techniques proposed by the research community are finding their way into products, and where software vendors tend to prefer versatile and robust techniques over more specialized ones, even though the latter perform factors better, we believe that the S-GRID is significant.

The key new benefit of the S-GRID is that it offers competitive query performance without making the assumption that all data points are known in advance, i.e., before the pre-computation can be accomplished in preparation for the processing. As another benefit, the S-GRID also enables query processing to proceed in parallel with updates to, and the consequent re-pre-computation on, the spatial network. Yet another benefit is that it is easy to integrate support for traffic regulations into the S-GRID approach.

Several directions for future work exist. First, it is of interest to perform analytical cost modeling of the S-GRID and to compare with the existing VN3, Island, and SPIE approaches. Second, a uniform two-dimensional grid has been used in the S-GRID. Since a non-uniform grid can capture more appropriately a network with dense and sparse regions, it is of interest to consider if or how a non-uniform grid can be used with the S-GRID approach. In addition, the partitioning can be made much more "network-aware" [6] in order to reduce the number of boundary points, which in turn may reduce the space consumption and the running time of the S-GRID approach. Third, this paper has hinted at how traffic regulations of real-world road networks can be accommodated in pre-computation. We believe, however, that real-world complexities such as those that stem from traffic regulations should be considered in more detail.

References

1. Cho, H.J., Chung, C.W.: An Efficient and Scalable Approach to CNN Queries in A Road Network. In: Proc. VLDB, pp. 865–876 (2005)
2. Hage, C., Jensen, C.S., Pedersen, T.B., Speicys, L., Timko, I.: Integrated Data Management for Mobile Services in the Real World. In: Proc. VLDB, pp. 1019–1030 (2003)
3. Huang, X., Jensen, C.S., Šaltenis, S.: The Islands Approach to Nearest Neighbor Querying in Spatial Networks. In: Bauzer Medeiros, C., Egenhofer, M.J., Bertino, E. (eds.) SSTD 2005. LNCS, vol. 3633, pp. 73–90. Springer, Heidelberg (2005)
4. Huang, X., Jensen, C.S., Šaltenis, S.: Multiple k Nearest Neighbor Query Processing in Spatial Network Databases. In: Manolopoulos, Y., Pokorný, J., Sellis, T. (eds.) ADBIS 2006. LNCS, vol. 4152, pp. 266–281. Springer, Heidelberg (2006)
5. Hu, H., Lee, D.L., Xu, J.: Fast Nearest Neighbor Search on Road Networks. In: Grust, T., Höpfner, H., Illarramendi, A., Jablonski, S., Mesiti, M., Müller, S., Patranjan, P.-L., Sattler, K.-U., Spiliopoulou, M., Wijsen, J. (eds.) EDBT 2006. LNCS, vol. 4254, pp. 186–203. Springer, Heidelberg (2006)
6. He, H., Wang, H., Yang, J., Yu., P.: BLINKS: Ranked Keyword Searches on Graphs. In: Proc. SIGMOD (to appear, 2007)
7. Jing, N., Huang, Y.W., Rundenstener, E.: Hierarchical Optimization of Optimal Path Finding for Transportation Applications. In: Proc. CIKM,, pp. 261–268 (1996)
8. Jensen, C.S, Kolář, J., Pedersen, T.B., Timko, I.: Nearest Neighbor Queries in Road Networks. In: Proc. ACM GIS,, pp. 1–8. ACM Press, New York (2003)
9. Kolahdouzan, M., Shahabi, C.: Voronoi-based Nearest Neighbor Search for Spatial Network Databases. In: Proc. VLDB,, pp. 840–851 (2004)
10. Kolahdouzan, M., Shahabi, C.: Alternative Solutions for Continuous k Nearest Neighbor Queries in Spatial Network Databases. GeoInformatica 9(4), 321–341 (2005)

11. Pearl, J.: Heuristics: Intelligent Search Strageties for Computer Problem Solving. Addison Wesley, Reading (1984)
12. Papadias, D., Zhang, J., Mamoulis, N., Tao, Y.: Query Processing in Spatial Network Databases. In: Proc. VLDB, pp. 802–813 (2003)
13. Shahabi, C., Kolahdouzan, M., Sharifzadeh, M.: A Road Network Embedding Technique for K-Nearest Neighbor Search in Moving Object Databases. GeoInformatica 7(3), 255–273 (2003)
14. Shekhar, S., Liu, D.: CCAM: A Connectivity-Clustered Access Method for Networks and Network Computations. TKDE 19(1), 102–119 (1997)
15. Xiong, X., Mokbel, M.F., Aref, W.G.: SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-Temporal Databases. In: Proc. ICDE, pp. 643–654 (2005)
16. Yiu, M.L., Mamoulis, N.: Clustering Objects on A Spatial Network. In: Proc. SIGMOD, pp. 443–454 (2004)
17. Yiu, M.L., Mamoulis, N., Papadias, D.: Aggregate Nearest Neighbor Queries in Road Networks. TKDE 17(6), 820–833 (2005)
18. Yiu, M.L., Papadias, D., Mamoulis, N., Tao, Y.: Reverse Nearest Neighbors in Large Graphs. TKDE 18(4), 540–553 (2006)